A new cohesion metric and restructuring technique for object oriented paradigm

Mehmet Kaya
Syracuse University

Jim Fawcett
Syracuse University, jfawcett@twcny.rr.com

SYR-EECS-2011-11 November 18, 2011

A New Cohesion Metric and Restructuring Technique for Object Oriented Paradigm

Mehmet Kaya mkaya@syr.edu
James W. Fawcett jfawcett@twcny.rr.com

ABSTRACT: When software systems grow large during maintenance, they may lose their quality and become complex to be read, understood and maintained. Developing a software system usually requires teams of developers working in concert to provide a finished product in a reasonable amount of time. What that means is many people may read each component of the software system such as a class in object oriented programming environment. We believe that a software component should be of good quality for the readers of the code to find its intents clear and the code behavior obvious. When this is the case it will be less costly to maintain the code and when its intent is clear, the code will be reusable, which is one of the key features of object oriented programming. Several software quality metrics have been proposed to measure overall or partial quality of software units such as classes or procedures. Cohesion is one of the most widely used metrics to measure quality of a software unit in terms of the relatedness of its components. This work presents a new cohesion metric based on program slicing and graph theory for units using object oriented paradigm. We believe that one can make a judgment on clarity of intent of the code using the metric we propose here. We aim to find out if a class is cohesive, handling one specific operation. We identify all program statements which constitute the operations in the same abstraction domain. When a class has more than one abstraction, this technique suggests a restructuring for generating more cohesive units based on this new cohesion metric.

KEYWORDS: Object Oriented Cohesion Metric, Code Restructuring, Extract Class, Program Slicing, Graph Theory

A New Cohesion Metric and Restructuring Technique for Object Oriented Paradigm

Mehmet Kaya
Department of Computer Science and Electrical Engineering
Syracuse University
Syracuse, NY, USA
mkaya@syr.edu

James W. Fawcett
Department of Computer Science and Electrical Engineering
Syracuse University
Syracuse, NY, USA
jfawcett@twcny.rr.com

***Abstract*— When software systems grow large during maintenance, they may lose their quality and become complex to be read, understood and maintained. Developing a software system usually requires teams of developers working in concert to provide a finished product in a reasonable amount of time. What that means is many people may read each component of the software system such as a class in object oriented programming environment. We believe that a software component should be of good quality for the readers of the code to find its intents clear and the code behavior obvious. When this is the case it will be less costly to maintain the code and when its intent is clear, the code will be reusable, which is one of the key features of object oriented programming. Several software quality metrics have been proposed to measure overall or partial quality of software units such as classes or procedures. Cohesion is one of the most widely used metrics to measure quality of a software unit in terms of the relatedness of its components. This work presents a new cohesion metric based on program slicing and graph theory for units using object oriented paradigm. We believe that one can make a judgment on clarity of intent of the code using the metric we propose here. We aim to find out if a class is cohesive, handling one specific operation. We identify all program statements which constitute the operations in the same abstraction domain. When a class has more than one abstraction, this technique suggests a restructuring for generating more cohesive units based on this new cohesion metric.**

Keywords- Object Oriented Cohesion Metric; Code Restructuring; Extract Class; Program Slicing; Graph Theory;

I. INTRODUCTION

Quality of a software unit is often measured with metrics like cohesion and coupling. Software developers put a great deal of effort into implementing high quality software units. High quality software has to be cohesive, readable, understandable, and reusable. Although these metrics can be used to measure the quality of a fragment of source code, in this work a unit refers to a class in an object oriented programming environment. In general, while cohesion metrics refer to the inter relatedness of a class's components, coupling measures the dependencies of a class with other units in the whole system.

Software development is a continuing process which requires maintenance after release of the software product. During this maintenance phase, developers may need to add some new functionality to the product or may need to change some of the existing functionalities based on changing needs of the users and transient working environments of the product. During this phase, changes made on the source code may reduce the quality of the software and make future changes more costly. Restructuring techniques help source code regain its quality after maintenance operations or sometimes even before the first release of the product.

A. Basics of Program Slicing

Program slicing is one of the preferred techniques to measure the cohesion level of software units. The concept of program slicing was first proposed by Mark Weiser [9, 12, and 13]. Weiser describes program slicing as the method of automatically decomposing programs by analyzing their relationships between statements based on data and control flow. Given the criterion C=(s, V), where s is a program statement and V is a subset of variables in a program P, program slicing is the process of finding all the program statements that affects value of a variable v in statement s. Figure 2 shows the program slice on the example program fragment P given in Figure 1, with respect to the criterion C= (9, sum). Here the number 9 represents the statement in line 9 of program P, i.e. cout << sum;

```
 1  int i;
 2  int sum = 0;
 3  int product = 1;
 4  for(i = 0; i < N; ++i)
 5  {
 6    sum = sum + i;
 7    product = product *i;
 8  }
 9  cout<< sum;
10  cout<< product;
```

Figure 1. Example Program Fragment P

Program slicing has been used in procedural programming extensively since it was first proposed by Weiser. An empirical study of some slice-based cohesion metrics can be found in [5]. Slice based cohesion metrics have been proposed for measuring cohesion level of a procedure and used in various studies [1-13]. In [6], reviews on slice-based cohesion measures for procedural paradigm and object-oriented paradigm are discussed. Slice-based cohesion measures for object oriented software discussed in [6] is either an extension of functional cohesion measures [11] or uses data member-method interactions for measurement [18], and they are quite different than our approach described in this paper. Some studies using program slicing, suggest restructuring for procedures by defining the low cohesive parts of them to be extracted from the procedure. In [10], this technique is used to measure the cohesiveness of each statement in a procedure and to identify parts of the procedure that cause low cohesion for restructuring purposes. In [22], cohesion level between output variables of a function is determined and a graph (pair-wise cohesion graph) is generated to visualize this relationship. After removing all edges that represent a cohesion below a given threshold level, they suggest restructuring the function based on connected components in the graph. Although the approach in [22] seems to be similar to ours, we generate our graph in a completely different way seeking semantic relationships between statements in object-oriented programs and our graph represents semantic relationships between data members of a class rather than cohesion values between them. Moreover we suggest extracting new reusable and cohesive classes rather than just a number of functions based on the cohesion metric we define for classes in object oriented software.

```
int i;
int sum = 0;
for(i = 0; i < N; ++i)
{
  sum = sum + i;
}
cout<< sum;
```

Figure 2. Slice of P with resp. to C=(9,sum)

In [2], the notion of data slices is defined and used for measuring functional cohesion. For each output variable v, the data slice is defined as the set of data tokens (i.e. variable and constant definition and references). Therefore a change made on any of the data tokens in data slice of v will affect the value of output variable v at some point. Cohesion is measured based on the percentage of data tokens that appear in more than one data slice and the ones that appear in all the data slices. In this version of slicing, data tokens are the basic units rather than statements in the program.

B. *Object-Oriented Cohesion Metrics and Restructuring*

To our knowledge, there has not been any program slicing based cohesion measure suggested for restructuring classes into more cohesive abstractions while preserving behavior of the whole system in object oriented environments. In our approach for measuring cohesion we use a graph representation of the slices found in a class for conceptual evaluation and for restructuring the class into a more cohesive form. This graph representation of program slices will be of great help to see which parts of the class are more related to each other at the statement level.

Although there has been very few slice based cohesion metrics proposed for measuring the quality of classes in object oriented paradigm, to our knowledge none has been proposed for restructuring purposes. In [11], slice based data cohesion measures for object oriented designs are defined as a modification of slice based functional cohesion measures defined by Bieman and Ott [2]. In [11], private and protected member variables of a class are considered as the data tokens and data slices are determined based on them. They do not alter the definition of measuring cohesion and use exactly the same measurement technique proposed by [2]. We believe that, although this study is one of the very few that use program slicing in object oriented paradigm, it does not support restructuring a class into more cohesive form once low cohesion level of the class is determined.

There are some other techniques used to indicate the cohesion level of a class. The majority of them use interactions between data members and methods of the class such as data member usage or sharing of data members [14, 15, 16, 17, 18, and 19]. Some studies represent this method-data member relationship with an undirected graph and suggest that the number of connected components in the graph indicates the cohesion level [17]. In [19] the dependency relationship between data members is also considered when evaluating the usage of a data member by a method. Although we believe that using graph theory in this manner is an important technique, none of these works suggest any restructuring to improve the cohesion level of a class once they quantitatively identified the low cohesion level of the class at hand.

In this work our evaluation of cohesion is quite different as we take executable statements and data members of a class and the relationship between them as the entities from which we construct our cohesion metrics. Therefore we believe that the information we get from that captures the semantic relationship of the data members more precisely than other techniques which consider only usage of data members in methods and method invocations.

Clustering Techniques is another preferred area of study for cohesion measurement and restructuring in object oriented paradigm. The basic idea in this technique is to group entities in a system (these entities are usually data members, methods and classes) based on the similarities between them or the relationships that they preserve in order to construct more cohesive groups [20].

Almost all applications of restructuring which use clustering techniques have this idea of moving data members or methods or some other entities that they define from one component to another. We believe that this approach may not be very useful for some cases as a method itself may not be cohesive. Our approach on the other hand, will help to generate classes with highly cohesive methods with the selected statements as it focuses on statements, data members, and semantic relationships between them, rather than just usage of data members in other components. In conclusion program slicing in this sense would be the most appropriate technique for restructuring classes as this technique reveals the actual relationship between components of interest.

C. *Paper Organization*

The rest of the paper is structured as follows. In section 2 we explain how we apply program slicing to object oriented classes with some new definitions of dependencies between statements. The section highlights the process of determination of slicing criteria and program slices based on the criteria. Section 3 introduces the Data-Slice-Graph (DSG) and the cohesion metrics we develop for it. A restructuring process based on the cohesion and DSG is given in section 4. Section 5 provides an application of the approach on a small explanatory source code. Concluding remarks are provided in Section 6.

II. Determination of Slicing Criteria and Construction of Program Slices

In this section we describe our slicing criteria and how we approach the identification process of program slices based on those criteria.

A. *Slicing Criteria in a Class*

The first step in our approach is to determine the slicing criteria in the class. To identify slicing criteria and slices in a class C, we have defined the following sets:

- DM_C is the set of all private data members defined in class C.
- ST_{dxC} is the set of all program statements which use data member d in C where d Є DM_C.

Therefore each element of the set ST_{dxC} represents a slicing criterion for data member d. Furthermore, for each d1 in DM_C, we have a corresponding set ST_{d1xC} representing the slicing criteria for data member d1.

B. *Dependency Relation between Statements*

In this study we use control and data dependencies between program statements to identify the slices in our target class. Informally speaking, a slice is the combination of backward and forward slices based on a criterion defined in above section. While a backward slice is defined as all the statements that the computation at the slicing criteria may depend on, forward slice consists of all the statements computation of which may be affected by the slicing criteria [23].

In this study, we analyze the dependencies between statements to construct the slices. Our primary focus is to find program statements which constitute operations in the same abstraction domain. For this reason we define a set of conditions for the statements to be evaluated in this manner. We say that two statements, S1 and S2, are dependent when one of the following conditions is true:

1. Execution of statement S1 is controlled by statement S2, or vice versa. An “if” control statement or a “for” loop statement is a good example for this case.
2. A variable defined in S1 is being used in S2
3. A variable, defined in statement S’ which uses a variable defined in S1, is being used in S2.
4. A variable defined in statement S’ is being used in both S1 and S2.
5. Invocation of a function *f ()* which includes the statement S1 is controlled by statement S2.
6. Execution of both S1 and S2 is controlled by the statement S’.
7. A variable defined in S1 is passed to a function *f* as an argument and the argument is being used in statement S2 of function f.

```
1   int i;
2   int sum = 0;
3   int product = 1;
4   for(i = 0; i < N; ++i)
5   {
6     sum = sum + i;
7     product = product *i;
8   }
9   cout<< sum;
10  cout<< product;
```
Figure 3. Slice of P with resp. to C=(9, sum)

Our definition of dependency relationship takes both direct and indirect dependencies into consideration and generates slightly different slices than other program slicing approaches do. After finding a specific slicing criterion, we take all the statements, which we think are semantically related to the criterion, into the slice rather than only the statements which affect value of a specific variable in the criterion. Figure 3 shows the slice we get from program fragment given in Figure 1 with respect to the criterion C=(9, sum) considering the dependency conditions stated above. Notice that from Figure 3 one can infer that all the statements are dependent in that program fragment with respect to the given criteria. We believe that in the scope of class cohesion and restructuring, considering all the dependency conditions listed above will lead to more accurate results as we seek semantic relationships between statements.

Figure 4 demonstrates the dependency conditions listed above using some fragments of the source code of the original version of *Class1* given in appendix. In condition 1, 5 and 6 of the figure, let the statements at line numbers 129, 131, 132 and 19 be represented by S’, S1, S2 and S3 respectively. S’, S1 and S2 are dependent since execution of S1 and S2 is controlled by S’ and moreover S’ and S3 are also dependent as invocation of function *ErrorInSize()*, which includes S3, depends on S’. In condition 2 and 3, let the statements at line numbers 64, 65, 66 and 67 be represented by S’, S1, S2 and S3 respectively. S’, S1, S2 and S3 are dependent because of their variable usage. In condition 4, let the statements at line numbers 111, 112 and 113 be represented by S’, S1 and S2 respectively. S’, S1 and S2 are dependent because a variable defined in S’ is used in both S1 and S2. In condition 7, let the statements at line numbers 29 and 101 be represented by S1 and S2 respectively. S1 and S2 are dependent because a variable defined in S1 is being passed to function *PushFunInvok(std::string str)* as an argument and the argument is being used in statement S2 of the function.

Condition 1, 5 and 6	
16	void ErrorInSize()
17	{
18	cout<<"Index out of range!\n";
19	cout<<"The Array has "<<top
20	<<" elements.\n";
21	}
129	if (top > 0)
130	{
131	top--;
132	int temp_int=stk[top];
133	return temp_int;
134	}
135	else
136	{
137	ErrorInSize();
138	return -1;
139	}
Conditions 2 and 3	
64	int w=x2-x1;
65	int h=y2-y1;
66	int a=w*h;
67	return a;
Condition 4	
111	string temp="Push invoked: ";
112	temp+=t;
113	PushFunInvok(temp);
Condition 7	
29	string temp="Class1 invoked: ";
30	temp+=t;
31	PushFunInvok(temp);
97	void PushFunInvok(std::string str)
98	{
99	if (topInvok < 100)
100	{
101	funinvokes[topInvok]=str;
102	topInvok++;
103	}
104	else
105	ErrorInSizeFunInvok();
106	}

Figure 4. Dependencies between Statements

C. *Determination of Program Slices*

To identify slices for data members defined in a class C, we have defined the following sets in addition to the sets defined in section 2.A:

- SL_{stxC} is the set of all program statements which directly or indirectly depend on the statement *st* based on the conditions listed in section 2.B. In other words, SL_{stxC} is the union of backward and forward slices based on the criterion of statement *st*.

- SL_{dxC} is the union of all SL_{stxC} where $st \in ST_{dxC}$ and $d \in DM_C$.

$$SL_{dxC} = \bigcup_{st \in ST_{dxC}} SLstxC$$

Therefore SL_{dxC} is the slice in our class C which includes all statements that directly or indirectly depend on at least one of the statements which reference data member d in C.

III. Data-Slice-Graph and a New Cohesion Metric

Class structure is the key unit of object oriented programming. Therefore, developers aim to design classes with high quality so that they can be reused, maintained, and tested easily. To reduce maintenance cost, these key units are expected to be simple, understandable, and readable as well.

Cohesion metrics have been studied extensively for the purpose of evaluating the relatedness of the components of a class. Different techniques have been used to quantify this aspect of the quality for a class. Most of the cohesion metrics are not proposed for restructuring classes to improve cohesion; therefore they may not be so practical to be used as a restructuring criterion. We believe that a cohesion metric based on program slicing idea will give more accurate results in object oriented environment as a restructuring criterion.

In object-oriented programming, a class generally is designed to handle one certain operation (one abstraction). To achieve this, most classes have some data members and functions to handle some part of the operation based on the clients' requests using some of the data members defined in the class. From this point we think that there is likely to be a relationship between the data members which are used to perform the intended operation of the class.

As software is exposed to changes during maintenance phase of software development life cycle, the classes may be altered in a way that they include some irrelevant components to their intended operations and therefore they no longer preserve their simplicity and reusability properties by including more than one abstraction. Or a class may initially be designed having more than one abstraction in it because of complex requirements or some other reasons. If the class has more than one abstraction, there must be a group of data members involving in each abstraction domain. In other words, if there is more than one independent group of data members in the class definition, we can say that there is more than one abstraction in the class and so it is not cohesive.

In this study, we aim to formalize the idea described in the paragraph above, and suggest a restructuring to partition the class into two or more cohesive target classes. For doing so, we generate data-slice-graphs (DSG) to visualize the relationship between the data members.

In some other works which use the idea of graph theory for measuring cohesion, they generate the graph based only on whether a method uses the data member of interest or not. A graph generated from this idea may not always reveal actual semantic relationship between data members. A graph generated from program slices, in our sense, will be of great help to see the real relationship between data members; hence we expect it will give more accurate results.

In DSG, each node represents a data member of the class which may possibly need to be restructured. We have the following definitions for DSG:

- DSG= (V, E) is a undirected graph such that V is the finite set of data members representing vertices in the graph and E is the finite set of relationships between data members representing edges in the graph.
- |V| is the number of data members of the class
- Let v1v2 represent an edge between two nodes v1 and v2;

$$v1v2 \in E \text{ iff } SL_{v1xC} \cap SL_{v2xC} \neq \emptyset$$

The description of DSG indicates that two data members, d1 and d2, are dependent if there is at least one program statement in the class that affects at least one occurrence of both data member d1 and data member d2 based on the dependency conditions given in section 2.B. Therefore the vertices, v1 and v2, representing data members d1 and d2 respectively, have an edge between them in DSG, i.e. v1v2 is in E.

We define the cohesion level of the class as the number of connected components, NC, in its DSG. The bigger NC the less cohesive our class is. Each connected component in DSG refers to one abstraction that the class holds.

IV. RESTRUCTURING THROUGH DSG

To propose restructuring for the class at hand, we use DSG and the number of connected components (NC) in DSG. Before discussing restructuring we shall explain what various values of NC mean:

- NC = 0 means there is not any data members defined in the class. That is a class that has no state - a utility class may be an example for this. Note that we do not apply this restructuring idea on this type of classes as DSG does not reveal any relationship for this kind of classes.
- NC = 1 occurs when the class has only one abstraction and when it is most cohesive. We do not restructure this kind of classes as this is the best situation a class may be in.
- NC > 1 occurs when the class has more than one abstraction. DSG reveals this by having more than one connected component and each connected component in this case represent one different abstraction the class is designed to handle. We restructure the code in this case and generate one cohesive class out of each connected component in DSG.

In DSG each connected component is a candidate to be extracted as a new smaller yet more cohesive class. In the example DSG given in Figure 5, C1 and C2 represent two different abstractions and our approach suggests that they should be extracted as new classes. Therefore data members represented by v1-v5 together with their slices are to be one class and data members represented by v6-v8 together with their slices are to be another class.

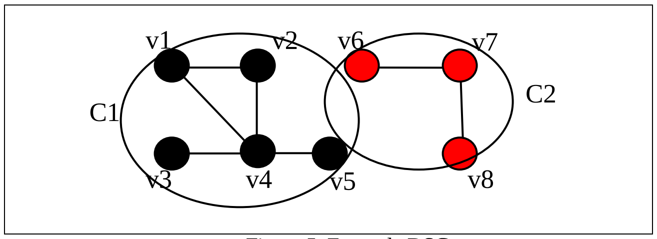

Figure 5. Example DSG

Our approach proposes this restructuring by defining independent sets of statements to be components of new classes. We propose to generate a method in the class with each consecutive set of statements in the slice of any data member that construct the connected component of the class. For example in Figure 5 data members v6, v7, and v8 construct the connected component for a new class C2 and each consecutive set of statements in the slices of these data members will be a method in C2.

In this study, we aim to come up with a technique to generate new cohesive and reusable classes from an existing less cohesive class. Although our approach fully defines all the statements related to one specific abstraction, restructuring the initial class to achieve our goal may require some simple analysis on the slices defined by our technique. In most cases extracting statements in the slices as methods into the new class is straightforward. Yet, there might be a few edge cases to handle for preventing any undesirable results. For example, we do not want to have any dependency from the classes we generate to the original class as this will affect the reusability of the classes by having a mutual dependency with the original class. This scenario is possible if a slice is including a function call. We have defined the following cases regarding possible problems with function calls during restructuring:

- Case 1: Function call in a control block

Definition: Our technique always guarantees that the function definition and the function call in this case are in the same slice. The 5th dependency condition listed in section 2.B assures this. Code fragment given for this condition in Figure 4 demonstrates this case. In that example code, statements at lines 18, 19, 20 and 137 are always guaranteed to be in the same slice.

Action: We suggest changing that function call in the control block with a function call to the corresponding function created in the new class. That will eliminate a call-back to the original class. In other words, during restructuring of the code fragment given in Figure 4, statements at line numbers 19, 20 and 21 will be moved into a new class as a function and statements at line numbers between 129 and 139 will be moved to the same class as another function. Let f1 and f2 represent these two functions respectively. We can assure this as all of the statements above will always reside in the same slice. Therefore we replace the function call *ErrorInSize();* in function f2 with a function call to function f1.

- Case 2: Function call without an argument

Definition: Our technique always guarantees that the function call in this case will not reside in any of the slices defined by our technique as there is not any data being used in this function call.

Action: We do not need to do anything special for this case. This case does not cause any call back to the original class.

- Case 3: Function call with an argument

Definition: Our technique does not guarantee that all of the statements in the definition of the function will always be in the same slice as function call, although we think that this is a case unlikely to happen. Yet, at least some parts of the function will be in the same slice with the function call, but the action in case 1 would not solve this problem in this case. The 7th dependency condition listed in section 2.B is related to this case and code fragment given for this condition in Figure 4 demonstrates it. In that code fragment function call at line number 31 passes an argument to the function *PushFunInvok(std::string str)* and statement at line 101 uses that argument. When we analyze this function we will see that all the statements in its definition are fully dependent with respect to the criterion C= (101, str) based on our dependency conditions. This makes the case to be just like case 1 and action in case 1 will handle any possible call backs for our example program.

Action: One should verify if the definition of the function consists of statements which are fully dependent based on some previously defined criteria or based on criterion of the statement which uses the argument passed to the function. If the function definition consists of fully dependent statements then this case is same as case 1. Otherwise this function call should be removed from the slice and the argument in the function call should be replaced with a return value from the function created in the new class corresponding to this consecutive set of statements where the function call resides.

Considering all of these cases will eliminate any possible undesirable dependencies that might arise from call-backs to the original class.

V. CASE STUDY

We now present an example for demonstration of this new cohesion and restructuring approach. We have our initial class as shown in appendix with the name of *Class1*. This class has 9 private data members and is not very well designed. Figure 6 shows all the properties of the class based on our approach for data members *stk* and *top*. Note that in the figure all the numbers given as an element of a set are the line numbers of the statements in the program. Sets for other data members of the class and their pair wise comparisons can be found in [21].

DM_{Class1} = {stk, top, funinvokes, topInvok, rawtime, x1, y1, x2, y2}
stk
$ST_{stkxClass1}$ ={116, 132} $SL_{116xClass1}$ ={114, 116, 117, 120, 18, 19} $SL_{132xClass1}$ ={129, 131, 132, 133, 137, 138, 18, 19} $SL_{stkxClass1} = SL_{116xClass1} \cup SL_{132xClass1}$ $SL_{stkxClass1}$ ={114,116,117,120,18,19,129, 131,132,133,137,138}
top
$ST_{topxClass1}$ ={19,32,87,114,116,117,129, 131,132,148} $SL_{19xClass1}$ ={19} $SL_{32xClass1}$ ={32} $SL_{87xClass1}$ ={87} $SL_{114xClass1}$ ={114,116,117,120,18,19} $SL_{116xClass1}$ ={114,116,117,120,18,19} $SL_{117xClass1}$ ={114,116,117,120,18,19} $SL_{129xClass1}$ ={129,131,132,133,137,18,19,138} $SL_{131xClass1}$ ={129,131,132,133,137,18,19,138} $SL_{132xClass1}$ ={129,131,132,133,137,18,19,138} $SL_{148xClass1}$ ={148} $SL_{topxClass1} = \bigcup_{st \in STtopxClass1} SLstxClass1$ = {19,32,87,114,116,117,120,18, 129,131,132,133,137,138,148}

Figure 6. Example Calculations for Class1

In Figure 6, the set $ST_{stkxClass1}$ is the set of all statements that use data member *stk* of *Class1*. This set represents the slicing criteria for that data member. In this case there are two statements which use data member *stk* in *Class1*, and they are the statements at line number 116 and 132. The sets $SL_{116xClass1}$ and $SL_{132xClass1}$ represent the slices based on the criterion of the statement at line number 116 and the criterion of the statement at line number 132 respectively. And finally the set $SL_{stkxClass1}$ represent the slice of data member *stk* including all the statements that is dependent on at least one occurrence of that data member.

After finding all the properties for each data member of the class as shown in Figure 6, now we are ready to construct DSG for the class. Figure 7 shows the process of constructing the graph.

Number of vertices
|V|=#of private data members=9
Edges
Let v1 and v2 represent data members *stk* and *top* respectively. $SL_{stkxClass1} \cap SL_{topxClass1}$={114,116,117,120,18,19,129, 131,132,133,137,138} $SL_{stkxClass1} \cap SL_{topxClass1} \neq \emptyset$ Therefore; $v1v2 \in E$

Figure 7. Construction Process of DSG

We have analyzed the codes for the given example class in appendix and generated the table shown in Figure 8 that demonstrates the pair-wise comparisons of the slices for each data member of the class.

Figure 8 shows the intersections of slices of pairs of data members of *Class1* given in appendix. "Ø" means that the intersection of the slices is the empty set for the two data members and ∩ means that there are some elements in the intersection of the slices. From this table, we generated the DSG in Figure 9.

∩	stk	top	funinvokes	topInvok	rawtime	x1	y1	x2	y2
stk	∩	∩	Ø	Ø	Ø	Ø	Ø	Ø	Ø
top	∩	∩	Ø	Ø	Ø	Ø	Ø	Ø	Ø
funinvokes	Ø	Ø	∩	∩	∩	Ø	Ø	Ø	Ø
topInvok	Ø	Ø	∩	∩	∩	Ø	Ø	Ø	Ø
rawtime	Ø	Ø	∩	∩	∩	Ø	Ø	Ø	Ø
x1	Ø	Ø	Ø	Ø	Ø	∩	∩	∩	∩
y1	Ø	Ø	Ø	Ø	Ø	∩	∩	∩	∩
x2	Ø	Ø	Ø	Ø	Ø	∩	∩	∩	∩
y2	Ø	Ø	Ø	Ø	Ø	∩	∩	∩	∩

Figure 8. Intersections of the Slices for Class1.

In Figure 9, we represent the relationship between data members shown in Figure 8 with an undirected graph representing the corresponding DSG for our class. Each connected component in the DSG in Figure 9 is shown with a different line style. In this case our corresponding DSG has three connected components. That means that the class we have analyzed has three abstractions in it and it is not cohesive and thus it should be restructured.

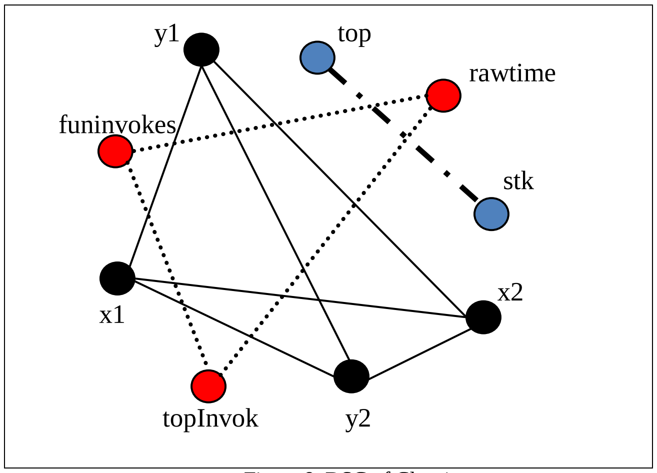

Figure 9. DSG of Class1

Our restructuring approach suggests creating a new class for each one of the connected component, found in DSG. In this case we suggest extracting three classes. The first class will include data members of *funinvokes*, *rawtime*, *topInvok* and their corresponding slices. The second class will include data members of *top* and *stk* and their corresponding slices. And lastly the third class will include *x1*, *x2*, *y1*, *y2* and their corresponding slices.

After extracting the suggested classes, the initial class will also be modified as it will invoke the proper functions of the new classes in the places where statements were removed. Therefore, since our approach does not suggest altering the public interface of the original class, this restructuring will not affect clients' code at all. The original and restructured versions of the example class *Class1* and the new classes that we extracted from original class are shown in the appendix. Because of the limitation on the number of pages of this paper we show the detailed process of restructuring in [21].

VI. Conclusion

In this study we have proposed a new cohesion metrics and an extract class restructuring technique for classes in object oriented environments using program slicing and graph theory. Our approach is different from other related works in a way that we try to find statements that constitutes the same abstraction in a class rather than regrouping existing components of a system. A tool support is needed for this approach to be applied to large software system and that remains as a future work of our study.

References

[1] M. Harman, S. Danicic, B. Sivagurunathan, B. Jones, and Y. Sivagurunathan. "Cohesion metrics," In Proceedings of the 8th International Quality Week, San Francisco CA, May 1995.

[2] J. M. Bieman, and L. M. Ott, "Measuring functional cohesion," IEEE Trans. Softw. Eng., vol. 20, no. 8, pp. 644–657, Aug 1994.

[3] S. Karstu, "An examination of the behavior of slice based cohesion measures," Master's thesis, Department of Computer Science, Michigan Technological University, 1994.

[4] H. D. Longworth, "Slice based program metrics," Master's thesis, Michigan Technological University, 1985.

[5] T. Meyers, and D. W. Binkley, "Slice-based cohesion metrics and software intervention," In 11th IEEE Working Conference on Reverse Engineering (WCRE 2004), pp. 256–265, 8-12 Nov 2004

[6] L. M. Ott, and J. M. Bieman, "Program slices as an abstraction for cohesion measurement," Information and Software Technology, vol. 40, no. 11/12, pp. 691–700, 1998.

[7] L. M. Ott, and J. J. Thuss, "The relationship between slices and module cohesion," In Proceedings of the 11th ACM conference on Software Engineering, pp.198–204, 1989.

[8] L. M. Ott, and J. J. Thuss, "Slice based metrics for estimating cohesion," In Proceedings of the IEEE-CS International Software Metrics Symposium, pp. 71–81, 21-22 May 1993.

[9] M. Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," PhD thesis, University of Michigan, Ann Arbor, 1979.

[10] J. Krinke, "Statement-Level Cohesion Metrics and their Visualization," Seventh IEEE Int. Working conference on Source Code Analysis and Manipulation, pp. 37-48, Sept. 30 2007-Oct. 1 2007

[11] B. Gupta, "A critique of cohesion measures in the object oriented paradigm," Master's thesis, Department of Computer Science, Michigan Technological University, 1997.

[12] M. Weiser, "Program Slicing," IEEE Transactions on Software Engineering, vol. 10, no. 4, pp. 352-357, July 1984

[13] M. Weiser, "Program Slicing," Proceedings of the 5th International Conference on Software Engineering, pp. 439-449, 1981

[14] S. R. Chidamber, and C. F. Kemerer, "Towards a Metrics Suite for Object-Oriented Design," Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, Vol. 26, No. 10, pp. 197-211, October 1991.

[15] S. R. Chidamber, and C. F. Kemerer, "A Metrics suite for object Oriented Design," IEEE Transactions on Software Engineering, Vol. 20, No. 6, pp. 476-493, June 1994.

[16] W. Li, and S. Henry, “Object oriented metrics that predict maintainability,” Journal of Systems and Software, vol. 23, no. 2, pp. 111-122, February 1993.

[17] M. Hitz, and B. Montazeri, “Measuring coupling and cohesion in object oriented systems,” Proceedings of the Int. Symposium on Applied Corporate Computing, October 1995.

[18] J. M. Bieman and B.-K. Kang, “Cohesion and Reuse in an Object-Oriented System,” Proc. Symp. Software Reusability, pp. 259-262, 1995.

[19] H. S. Chae, Y. R. Kwon, and D. H. Bae, “Improving Cohesion Metrics for Classes by Considering Dependent Instance Variables,” IEEE Transaction on Software Engineering, vol. 30, no. 11, pp. 826-832, November 2004,

[20] G. Şerban, I-G. Czibula, “Restructuring Software Systems Using Clustering,” 22nd Int. Symp. on Computer and Information Sciences (ISCIS), pp. 1-6, 7-9 Nov 2007

[21] All the example codes and detailed restructuring process is posted on “http://www.ecs.syr.edu/faculty/fawcett/handouts/research/kaya/files/appendix.pdf”

[22] A. Lakhotia, and J.-C. Deprez, “Restructuring Functions with Low Cohesion,” Proc. Sixth Working Conf. Reverse Eng. (WCRE), pp. 36-46, 6-8 Oct 1999.

[23] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” ACM Trans. Programming Lang. and Syst. (TOPLAS),vol. 12, no. 1, pp. 26-60, Jan. 1990

APPENDIX

Original Class - Class1

```
class Class1
{
private:
  int stk[100];
  int top;
  string funinvokes[100];
  int topInvok;
  time_t rawtime;
  int x1, y1, x2, y2;
  void ErrorInSizeFunInvok()
  {
    cout<<"Index out of range!\n";
    cout<<"The Array has "<<topInvok
           <<" elements.\n";
  }
  void ErrorInSize()
  {
    cout<<"Index out of range!\n";
    cout<<"The Array has "<<top
           <<" elements.\n";
  }
public:
  Class1(int left=0,int up=0,
         int right=0,int bottom=0)
  {
    topInvok=0;
    time ( &rawtime );
    string t=string(ctime(&rawtime));
    string temp="Class1 invoked: ";
    temp+=t;
    PushFunInvok(temp);
    top=0;
    x1=left;
    y1=up;
    x2=right;
    y2=bottom;
  }
  ~Class1() {}
  int Height()
  {
    time ( &rawtime );
    string t=string(ctime(&rawtime));
    string temp="Height invoked: ";
    temp+=t;
    PushFunInvok(temp);
    return (y2-y1);
  }
  int Width()
  {
    time ( &rawtime );
    string t=string(ctime(&rawtime));
    string temp="Width invoked: ";
    temp+=t;
    PushFunInvok(temp);
    return (x2-x1);
  }
  int Area()
  {
    time ( &rawtime );
    string t=string(ctime(&rawtime));
    string temp="Area invoked: ";
    temp+=t;
    PushFunInvok(temp);
    int w=x2-x1;
    int h=y2-y1;
    int a=w*h;
    return a;
  }
  int Perimeter()
  {
    time ( &rawtime );
    string t=string(ctime(&rawtime));
    string temp="Perimeter invoked:";
    temp+=t;
    PushFunInvok(temp);
    int w=x2-x1;
    int h=y2-y1;
    return 2*w+2*h;
  }
  void Clear()
  {
    time ( &rawtime );
    string t=string(ctime(&rawtime));
    string temp="Clear invoked: ";
    temp+=t;
    PushFunInvok(temp);
    top=0;
  }
  void printAllInvoks()
  {
    for(int i=0; i<topInvok; i++)
    {
      string temp=funinvokes[i];
      cout<<temp;
    }
  }
  void PushFunInvok(std::string str)
  {
    if (topInvok < 100)
    {
        funinvokes[topInvok]=str;
        topInvok++;
    }
    else
       ErrorInSizeFunInvok();
  }
  void Push(int i)
  {
    time ( &rawtime );
    string t=string(ctime(&rawtime));
    string temp="Push invoked: ";
    temp+=t;
    PushFunInvok(temp);
    if (top < 100)
    {
        stk[top]=i;
        top++;
    }
    else
        ErrorInSize();
  }
  int Pop()
  {
    time ( &rawtime );
    string t=string(ctime(&rawtime));
    string temp="Pop invoked: ";
    temp+=t;
    PushFunInvok(temp);
    if (top > 0)
    {
           top--;
           int temp_int=stk[top];
           return temp_int;
    }
    else
    {
           ErrorInSize();
           return -1;
    }
  }
  int Size()
  {
    time ( &rawtime );
    string t=string(ctime(&rawtime));
    string temp="Size invoked: ";
    temp+=t;
    PushFunInvok(temp);
    return top;
  }
};
```

Restructured Version of Original Class - Class1

```
class Class1
{
 private:
  New1* n1;
  New2* n2;
  New3* n3;
  void ErrorInSizeFunInvok()
  {
      n2->fun2_1();
  }
  void ErrorInSize()
  {
      n1->fun1_1();
  }
 public:
  Class1(int left=0,int up=0,
         int right=0,int bottom=0)
  {
   n1=new New1();
   n2=new New2();
   n3=new New3(left,up,right,bottom);
  }
  ~Class1() {}
  int Height()
  {
    n2->fun2_2();
    return n3->fun3_1();
  }
  int Width()
  {
    n2->fun2_3();
    return n3->fun3_2();
  }
  int Area()
  {
    n2->fun2_4();
    return n3->fun3_3();
  }
  int Perimeter()
  {
    n2->fun2_5();
    return n3->fun3_4();
  }
  void Clear()
  {
    n2->fun2_6();
    n1->fun1_2();
  }
  void printAllInvoks()
  {
    n2->fun2_7();
  }
  void PushFunInvok(std::string str)
  {
    n2->fun2_8(str);
  }
  void Push(int i)
  {
    n2->fun2_9();
    n1->fun1_3(i);
  }
  int Pop()
  {
    n2->fun2_10();
    return n1->fun1_4();
  }
  int Size()
  {
    n2->fun2_11();
    return n1->fun1_5();
  }
};
```

Extracted Class 1

```
class New1
{
 private:
  int stk[100];
  int top;
 public:
  New1()
  {
    top=0;
  }
  void fun1_1()
  {
    cout<<"Index out of range!\n";
    cout<<"The Array has "<<top
            <<" elements.\n";
  }
  void fun1_2()
  {
                top=0;
  }
  void fun1_3(int i)
  {
                if (top < 100)
                {
                            stk[top]=i;
                            top++;
                }
                else
                            fun1_1();
  }
  int fun1_4()
  {
    if (top > 0)
    {
                top--;
                int temp_int=stk[top];
                return temp_int;
    }
    else
    {
                fun1_1();
                return -1;
    }
  }
  int fun1_5()
  {
    return top;
  }
};
```

Extracted Class 2

```
class New2
{
 private:
   string funinvokes[100];
   int topInvok;
   time_t rawtime;
 public:
 New2()
 {
   topInvok=0;
   time ( &rawtime );
   string t=string(ctime(&rawtime));
   string temp="Class1 invoked: ";
   temp+=t;
   fun2_8(temp);
 }
 void fun2_1()
 {
    cout<<"Index out of range!\n";
    cout<<"The Array has "<<topInvok
        <<" elements.\n";
 }
 void fun2_2()
 {
   time ( &rawtime );
   string t=string(ctime(&rawtime));
   string temp="Height invoked: ";
   temp+=t;
   fun2_8(temp);
 }
 void fun2_3()
 {
   time ( &rawtime );
   string t=string(ctime(&rawtime));
   string temp="Width invoked: ";
   temp+=t;
   fun2_8(temp);
 }
 void fun2_4()
 {
   time ( &rawtime );
   string t=string(ctime(&rawtime));
   string temp="Area invoked: ";
   temp+=t;
   fun2_8(temp);
 }
 void fun2_5()
 {
   time ( &rawtime );
   string t=string(ctime(&rawtime));
   string temp="Perimeter invoked:";
   temp+=t;
   fun2_8(temp);
 }
 void fun2_6()
 {
   time ( &rawtime );
   string t=string(ctime(&rawtime));
   string temp="Clear invoked: ";
   temp+=t;
   fun2_8(temp);
   }
 void fun2_7()
 {
   for(int i=0; i<topInvok; i++)
   {
          string temp=funinvokes[i];
          cout<<temp;
   }
 }
 void fun2_8(string str)
 {
   if (topInvok < 100)
   {
          funinvokes[topInvok]=str;
          topInvok++;
   }
   else
          fun2_1();
 }
 void fun2_9()
 {
   time ( &rawtime );
   string t=string(ctime(&rawtime));
   string temp="Push invoked: ";
   temp+=t;
   fun2_8(temp);
 }
 void fun2_10()
 {
   time ( &rawtime );
   string t=string(ctime(&rawtime));
   string temp="Pop invoked: ";
   temp+=t;
   fun2_8(temp);
 }
 void fun2_11()
 {
   time ( &rawtime );
   string t=string(ctime(&rawtime));
   string temp="Size invoked: ";
   temp+=t;
   fun2_8(temp);
 }
};
```

Extracted Class 3

```
class New3
{
 private:
   int x1, y1, x2, y2;
 public:
 New3(int left,int up,
      int right,int bottom)
 {
   x1=left;
   y1=up;
   x2=right;
   y2=bottom;
 }
 int fun3_1()
 {
   return (y2-y1);
 }
 int fun3_2()
 {
   return (x2-x1);
 }
 int fun3_3()
 {
   int w=x2-x1;
   int h=y2-y1;
   int a=w*h;
   return a;
 }
 int fun3_4()
 {
   int w=x2-x1;
   int h=y2-y1;
   return 2*w+2*h;
 }
};
```